BIAX Corporation v. Intel
Civil Action No. 2:05-cv-184-TJW

EXHIBIT 2
(PART 3)

analysis of an instruction stream to provide the additional intelligent information for routing and synchronization is performed only once during the program preparation process (i.e., static allocation) on a given piece of software, and is not performed during execution (i.e, dynamic allocation) as is found in some conventional prior art approaches. The analysis is performed on the object code output from conventional compilers and therefore is programming language independent.

2. General Software Description

In FIG. 1, the general description of the software of the present invention is set forth and is generally termed "TOLL." The software TOLL located in a processing system **160** operates on standard compiler output **100** which is typically object code or an intermediate object code such as "p-code." It is known that the output of conventional compilers is a sequential stream of object code instructions hereinafter referred to as the instruction stream. Conventional language processors typically perform the following functions in generating the sequential instruction stream:

1. lexical scan of the input text,
2. syntactical scan of the condensed input text including symbol table construction,
3. performance of machine independent optimization including parallelism detection and vectorization, and
4. an intermediate (PSEUDO) code generation including instruction functionality, resources required, and structural properties.

In the creation of the sequential instruction stream, the conventional compiler creates a series of basic blocks (BBs) which are single entry single exit (SESE) groups of contiguous instructions. See, for example, *Principles of Compiler Design,* Alfred v. Aho and Jeffery D. Ullman, Addison Wesley, 1979, pg. 6, 409, 412–413 and *Compiler Construction for Digital Computers,* David Gries, Wiley, 1971. The conventional compiler, although it utilizes basic block information in the performance of its tasks, provides an output stream of sequential instructions without any basic block designations. The TOLL software of the present invention is designed to operate on the formed basic blocks (BBs) which are created within a conventional compiler. In each of the conventional SESE basic blocks there is exactly one branch (at the end of the block) and there are no control dependencies; the only relevant dependencies within the block are those between the resources required by the instructions.

The output of the compiler **100** in the basic block format, is shown in FIG. 2. The TOLL software **110** of the present invention being processed in a computer **160** performs three basic determining functions on the compiler output **100** as shown in FIG. 1. These functions are to analyze the resource usage of the instructions **120**, extend intelligence for each instruction in each basic block **130**, and to build execution sets composed of one or more basic blocks **140**. The resulting output of these three basic functions **120, 130,** and **140** from processor **160** comprise the TOLL output **150** of the present invention.

At the outset, the TOLL software of the present invention operates on a compiler output **100** only once and without user intervention. Therefore, for any given program, the TOLL software need operate on the compiler output **100** only once.

The functions of the TOLL software **110** are to analyze the instruction stream in each basic block for natu-

ral concurrencies, to perform a translation of the instruction stream onto the actual hardware system of the present invention, to alleviate any hardware induced idiosyncracies that may result from this translation and to encode the resulting stream into an actual machine language to be used on the hardware of the present invention. The TOLL software **110** performs these features by analyzing the instruction stream and then assigning context free processor elements and resources as a result thereof. The TOLL software **110** provides the "synchronization" of the overall system by assigning appropriate firing times to each instruction in the instruction stream.

Instructions can be dependent on one another in a variety of ways although there are only three basic types of dependencies. First, there are procedural dependencies due to the actual structure of the instruction stream; i.e., instructions may follow one another in other than a sequential order due to branches, jumps, etc. Second, operational dependencies are due to the finite number of hardware elements present in the system. These hardware elements include the registers, condition codes, stack pointers, processor elements, and memory. Thus if two instructions are to execute in parallel, they must not require the same hardware element unless they are both reading that element (provided of course, that the element is capable of being read simultaneously). Finally, there are data dependencies between instructions in the instruction stream. This form of dependency will be discussed at length later. However, within a basic block only data and operational dependencies are present.

The TOLL software **110** maintains the proper execution of a program; i.e., TOLL must assure that the parallelized code **150** generates the same results as those of the original serial code. In order to do this, the parallelized code **150** must access the resources in the same relative sequence as the serial code for instructions that are dependent on one another; i.e., the relative ordering must be satisfied. However, independent sets of instructions may be effectively executed out of sequence.

In Table 1 is set forth an example of a SESE basic block representing the inner loop of a matrix multiply routine. This example will be used throughout this specification and the teachings of the present invention are application for any routine. The instruction is set forth in the left hand column and the conventional object code function for this basic block is represented in the right hand column.

TABLE 1

INSTRUCTION	OBJECT CODE
LD R0, (R10) +	I0
LD R1, (R11) +	I1
MM R0, R1, R2	I2
ADD R2, R3, R3	I3
DEC R4	I4
BRNZR LOOP	I5

The instruction stream contained within the SESE basic block set forth in Table 1 performs the following functions. In instruction I0, register R0 is loaded with the contents of memory whose address is contained in R10. The instruction shown above increments the contents of R10 after the address has been fetched from R10. The same statement can be made for instruction I1, with the exception that register R1 is loaded and register R11 incremented. The contents of register R0 are then multiplied with the contents of register R1 and stored in register R2 in instruction I2. In instruction I3, the contents of register R2 and register R3 are added together and stored in register R3. In instruction I4, register R4 is decremented. Note that instructions I2, I3 and I4 also generate a set of condition codes that reflect the status of their respective execution. In instruction I5, the contents of register R4 are indirectly tested for zero (via the condition codes generated by instruction I4). A branch occurs if the decrement operation produced a non-zero value; otherwise execution proceeds with the next basic block's first instruction.

As shown in FIG. 1, the first function performed by TOLL 110 is to analyze the resource usage of the instructions; which in Table 1 are instructions I0 through I5. The TOLL software 110 analyzes each instruction to ascertain the resource requirements of each instruction.

This analysis is important in determining whether or not sets of resources share any elements and, therefore, whether or not the instructions are independent of one another. Clearly, mutually independent instructions can be executed in parallel and are termed "naturally concurrent." Instructions that are independent can be executed in parallel and do not rely on one another for any information nor do they share any hardware resources in other than a read only manner.

On the other hand, instructions that are dependent on one another can be formed into a set wherein each instruction in the set is dependent on every other instruction in that set, although this dependency may not be direct. The set can be described by the instructions within that set, or conversely, by the resources used by the instructions in that set. Instructions within different sets are completely independent of one another, i.e., there are no resources shared by the sets. Hence, the sets are independent of one another.

In the example of Table 1, there are two independent sets of dependent instructions determined by TOLL:

Set 1:	CC1:	I0, I1, I2, I3 and
Set 2:	CC2:	I4, I5

As can be seen, instructions I4 and I5 are independent of instructions I0–I3. In set 2, I5 is directly dependent on I4. In set 1, I2 is directly dependent on I0 and I1. Instruction I3 is directly dependent on I2 and indirectly dependent on I0 and I1.

The TOLL software of the present invention detects these independent sets of dependent instructions and assigns condition codes sets such as CC1 and CC2 to each set. This avoids the operational dependency that would occur if only one set of condition codes were available to the instruction stream.

In other words, the results of the execution of instructions I0 and I1 are needed for the execution of instruction I2. Similarly, the results of the execution of instruction I2 are needed for the execution of instruction I3. Thus, the TOLL software 110 determines if an instruction will perform a read and/ or a write to a resource. This functionality is termed the resource requirement analysis of the instruction stream.

It should be noted that, unlike the teachings of prior art, the present invention teaches that it is not necessary for dependent instructions to execute on the same processor element. The determination of dependencies is

needed only to assign condition code sets and to assign instruction firing times, as will be described later. The present invention can execute dependent instructions on different processor elements because of the context free nature of the processor elements and the total coupling of the processor elements to the shared resources, such as register files, as will also be described later.

The results of the analysis stage 120, for the example set forth in Table 1, are set forth in Table 2.

TABLE 2

BB	FUNCTION
I0	Memory Read, Reg. Write, Reg. Read & Write
I1	Memory Read, Reg. Write, Reg. Read & write
I2	Two Reg. Reads, Reg. Write, Set Cond. Code (Set #1)
I3	Two Reg. Reads, Reg. Write, Set Cond. Code (Set #1)
I4	Read Reg., Reg. Write, Set Cond. Code (Set #2)
I5	Read Cond. Code (Set #2)

In Table 2, for instructions I0 and I1, a register is read and written followed by a memory read (at a distinct address), followed by a register write. Likewise, condition code writes and register reads and writes occur for instructions I2 through I4. Finally, instruction I5 is a simple read of a condition code storage and a resulting branch or loop.

The second pass 130 through the SESE basic block 100 is to add or extend the intelligence of each instruction within the basic block. This is the assignment of an instruction's execution time relative to the execution times of the other instructions in the stream, the assignment of a logical processor number on which the instruction is to execute and the assignment of any static shared context storage mapping information that may be needed.

In order to assign the instruction's firing time, the temporal usage of each resource required by the instruction must be considered. The temporal usage of each resource is characterized by a "free time" and a "load time." The free time is the last time the resource is used by an instruction. The load time is the last time the resource is modified by an instruction. If an instruction is going to modify a resource, it must execute after the last time the resource is used, in other words, after the free time. If an instruction is going to read the resource, it must perform the read after the last time the resource has been loaded, in other words, after the load time.

The link between the temporal usage of each resource and the actual usage of the resource is as follows. If the instruction is going to write/modify the resource, the last time the resource is read or written by other instructions (i.e., the "free time" for the resource) plus one time interval will be the firing time for this resource. The "plus one time interval" comes from the fact that an instruction is still using the resource during the free time. On the other hand, if the instruction reads a resource, the last time the resource is modified by other instructions (i.e., the load time for the resource) plus one time interval will be the resource firing time. The "plus one time interval" comes from the time required for the instruction that is performing the load to execute.

The above discussion assumes that the exact location of the resource that is accessed is known. This is always true of resources that are directly named such as registers and condition codes. However, memory operations may, in general, be to unknown (at compile time) locations. In particular, addresses that are generated by effective addressing constructs fall under this domain. In the previous example, it has been assumed (for the purposes of communicating the basic concepts of TOLL) that the addresses used by instructions **I0** and **I1** are distinct. If this were not the case, the TOLL software would assure that only those instructions that did not use memory would be allowed to execute in parallel with an instruction that was accessing an unknown location in memory.

The resource firing time is evaluated by TOLL **110** for each resource that the instruction uses. These resource firing times are then compared to determine which is the largest or latest with this maximum value determining the actual firing time assigned to the instruction. At this point, TOLL **110** then updates all resources' free and load times, to reflect this instruction's firing time. TOLL **110** then proceeds by analyzing the next instruction.

There are many methods available for determining interinstruction dependencies within a basic block. The previous discussion is just one possible implementation assuming a specific compiler-TOLL partitioning. Many other compiler-TOLL partitionings and methods for determining inter-instruction dependencies may be possible and realizable to one skilled in the art. As an example, the TOLL software uses a linked list analysis to represent the data dependencies within a basic block. Other possible data structures that could be used are trees, stacks, etc.

Assume a linked list representation is desired for the analysis and representation of the inter-instruction dependencies. Each register is associated with a set of pointers to the instructions that use the value contained in that register. For the matrix multiply example in Table **1**, the resource usage is set forth in the following:

TABLE 3

Resource	Loaded By	Read By
R0	I0	I2
R1	I1	I2
R2	I2	I3
R3	I3	I3, I2
R4	I4	I5
R10	I0	I0
R11	I1	I1

Thus, by following the "read by" links and knowing the resource utilization for each instruction, the independencies of Sets **1** and **2**, above, are constructed in the analyze instruction stage **120** by TOLL **110**.

For purposes of the example of Table 1, it is assumed that the basic block commences with an arbitrary time interval, in an instruction stream, such as time interval, for example purposes only, **T16**. In other words, this particular basic block in time sequence is assumed to start with time interval **T16**. The results of the analysis in stage **120** are set forth in Table 4.

TABLE 4

REG	I0	I1	I2	I3	I4	I5
R0	T16		T17			
R1		T16	T17			
R2			T17	T18		
R3				T18		
R4					T16	
CC1			T17	T18		
CC2						T17
R10	T16					
R11		T16				

The left hand column of Table 4 relates to the identity of the register or condition code storage whereas the rows in the table represent the instructions in the basic block example of Table **1**. Instruction **I0** requires that a register be read and written and another register written at time **T16**, the start of the basic block. Hence, at time **T16**, register **R10** is read from and written into and register **R0** is written into.

Under the teachings of the present invention, there is no reason that registers **R1**, **R11**, and **R4** cannot also have operations performed on them during time **T16**. These three instructions, **I0**, **I1**, and **I4**, are data independent of each other and can be operated on concurrently during time **T16**. Instruction **I2** requires first that registers **R0** and **R1** be operated on so that they may be multiplied together and the results stored in register **R2**. Although, register **R2** could be operated on in time **T16**, instruction **I2** is data dependent and depends upon the results of loading registers **R0** and **R1**, which occurs during time **T16**. Therefore, the completion of instruction **I2** is data dependent and must occur during or after timeframe **T17**. Hence, in Table **4** above, the entry **T17** for the intersection of instruction **I2** and register **R2** is underlined because it is data dependent. Likewise, instruction **I3** requires data to be present in register **R2** which first occurs during time **T17**. Hence, register **R2** can have an operation occur on it only during or after time **T18**. As it turns out, instruction **I5** depends upon the reading of the condition code storage **CC2** which is updated in instruction **I4**. The reading of the condition code storage is data dependent upon the results stored in time **T16** and, therefore, must occur during or after the next time, **T17**.

Hence, in stage **130**, the object code instructions are assigned "instruction firing times" (IFTs) as set forth in Table **5** based upon the above analysis.

TABLE 5

OBJECT CODE	INSTRUCTION FIRING TIME (IFT)
I0	T16
I1	T16
I2	T17
I3	T18
I4	T16
I5	T17

Each of the instructions in the sequential instruction stream in a basic block can be performed in the assigned time intervals. As is clear in Table **5**, the same six instructions of Table **1** normally processed sequentially in six cycles can be processed, under the teachings of the present invention, in only three firing times: **T16**, **T17**, and **T18**. The instruction firing time (IFT) provides the "time-driven" feature of the present invention.

The next function in the extend intelligence stage **130** is to reorder the natural concurrencies in the instruction stream according to instruction firing times (IFTs) and then to assign the individual logical parallel processors. It should be noted that the reordering is only required due to limitations in currently available technology. If true fully associative memories were available, the reordering of the stream would not be required and the processor numbers could be assigned in a first come, first served manner. The hardware of the instruction selection mechanism could be appropriately modified by one skilled in the art to address this mode of operation.

For example, assuming currently available technology, and a system with four parallel processor elements (PEs) and a branch execution unit (BEU) within each LRD, the processor elements and the branch execution unit can be assigned, under the teachings of the present invention, according to that set forth in Table 6 below. It should be noted that the processor elements execute all non-branch instructions, while the branch execution unit (BEU) of the present invention executes all branch instructions. These will be described in greater detail subsequently.

TABLE 6

Logical Processor Number	T16	T17	T18
0	I0	I2	I3
1	I1	—	
2	I4	—	—
4	—	—	—
BEU	—	I5(delay)	—

Hence, under the teachings of the present invention, during time interval T16, parallel processor elements 0, 1, and 2 concurrently process instructions I0, Il, and I4. Likewise, during the next time interval T17, parallel processor elements 0 and the user's BEU concurrently process instructions I2 and I5. And finally, during time interval T18, processor element 0 processes instruction I3. During instruction firing times T16, T17, and T18, parallel processor element 3 is not utilized in the example of Table 1. In actuality, since the last instruction is a branch instruction, the branch cannot occur until the last processing is finished in time T18 for instruction I3. A delay field is built into the processing of instruction I5 so that even though it is processed in time interval T17, its execution is delayed before looping or branching out until after instruction I3 has executed.

The TOLL software 110 of the present invention in the extend intelligence stage 130 looks at each individual instruction and its resource usage both as to type and as to location (if known) (e.g , Table 3). It then assigns instruction firing times (IFTs) on the basis of this resource usage (e.g., Table 4), reorders the instruction stream based upon these firing times (e.g., Table 5) and assigns logical processor numbers (LPNs) (e.g., Table 6) as a result thereof.

The extended intelligence information involving the logical processor number (LPN) and the instruction firing time (IFT) is added to each instruction of the basic block as shown in FIGS. 3 and 4. As will also be pointed out subsequently, this extended intelligence (EXT) for each instruction in a basic block (BB) will be translated onto the physical processor architecture of the present invention. The physical translation is done by hardware. It is important to note that the actual hardware may contain less, the same as, or more physical processor elements than the number of logical processor elements.

The Shared Context Storage Mapping (SCSM) information shown in FIG. 4 and attached to an instruction has two components, static and dynamic. Static information is attached by the TOLL software or compiler and is a result of the static analysis of the instruction stream. Dynamic information is attached at execution time by a logical resource drive (LRD) as will be discussed later.

At this stage 130, the TOLL software 110 has analyzed the instruction stream as a set of single entry single exit (SESE) basic blocks (BBs) for natural concurrencies that can be processed individually by separate processor elements (PEs) and has assigned to each instruction an instruction firing time (IFT) and a logical processor number (LPN). Under the teachings of the present invention, the instruction stream is pre-processed by TOLL to statically allocate all processing resources in advance of execution. This is done once for any given program and is applicable to any one of a number of different program languages such as FORTRAN, COBOL, PASCAL, BASIC, etc.

In stage 140, the TOLL software 110 builds execution sets (ESs). These are set forth in FIG. 5 wherein a series of basic blocks (BBs) form a single execution set (ES). Once TOLL identifies an execution set 500, header 510 and/or trailer 520 information is placed on the ends. In the preferred embodiment only header information 510 is attached although the invention is not so limited.

Under the teachings of the present invention, basic blocks generally follow one another in the instruction stream. There may be no need for re-ordering of the basic blocks even though individual instructions within a basic block, as discussed above, are re-ordered and assigned extended intelligence information. However, the invention is not so limited. Each basic block is single entry and single exit (SESE) with the exit through a branch instruction. Typically, the branch to another instruction is within a localized neighborhood such as within 400 instructions of the branch. The purpose of forming the execution sets (stage 140) is to determine the minimum number of basic blocks that can exist within an execution set such that the number of "instruction cache faults" are minimized. In other words, in a given execution set, branches or transfers out of an execution set are statistically minimized. TOLL in stage 140 can use a number of conventional techniques for solving this linear programming-like problem which is based upon branch distances and the like. The purpose is to define an execution set as set forth in FIG. 5 so that the execution set can be placed in a hardware cache, as will be discussed subsequently, in order to minimize instruction cache faults (i.e., transfers out of the execution set).

What has been set forth above is an example shown in Tables 1 through 6 of TOLL software 110 in a single context of use. In essence, TOLL determines the natural concurrencies within the instruction streams for each basic block within a given program. TOLL adds an instruction firing time (IFT) and a logical processor number (LPN) to each instruction in the determined natural concurrencies. Hence, all processing resources are statically allocated in advance of processing. The TOLL software of the present invention can be used in a number of different programs, each being used by the same or different users on a processing system of the present invention as will be explained next.

3. General Hardware Description

In FIG. 6, the block diagram format of the system architecture of the present invention termed "TDA" is shown. The TDA system architecture 600 includes a memory sub-system **610** interconnected to a number of logical resource drivers (LRDs) **620** over a network **630**. The logical resource drivers **620** are further interconnected to a group of context free processor elements **640** over a network **650**. Finally, the group of processor elements **640** are connected to the shared resources containing a pool of register set and condition code set files **660** over a network **670**. The LRD-memory network **630**, the PE-LRD network **650**, and the PE-context file network **670** are full access networks that could be composed of conventional crossbar networks, omega networks, banyon networks, or the like. The networks are full access (non-blocking in space) so that, for example, any processor element **640** can access any register file or condition code file in any context **660**. Likewise, any processor element **640** can access any logical resource driver **620** and any logical resource driver **620** can access any portion of the memory subsystem **610**. In addition, the PE-LRD and PE-context networks are non-blocking in time. In other words, these two networks guarantee access to any resource from any resource regardless of load conditions on the network. The architecture of the switching elements of the PE-LRD network **650** and the PE-context network **670** are considerably simplified since the TOLL software guarantees that collisions in the network will never occur. The diagram of FIG. **6** is a MIMD system wherein one context **660** corresponds to at least one user program.

The memory subsystem **610** can be constructed using a conventional memory architecture and conventional memory elements. There are many such architectures and elements that could be constructed by a person skilled in the art that would satisfy the requirements of this system. For example, a banked memory architecture could be used. *High Speed Memory Systems*, A.V. Pohm and O.P. Agrawal, Reston Publishing Co., 1983.

The logical resource drivers **620** are unique to the system architecture **600** of the present invention. Each LRD provides the data cache and instruction selection support for a single user (who is assigned a context) on a timeshared basis. The LRDs receive execution sets from the various users wherein one or more execution sets per context is stored on any given LRD. The instructions within the basic blocks of the stored execution sets are stored in queues based on the logical processor number. For example, if the system has 64 users and 8 LRDs, 8 users would share an individual LRD on a timeshared basis. The operating system determines who gets an individual LRD and for how long. The LRD is detailed at length subsequently.

The context free processor elements **640** are also unique to the TDA system architecture and will be discussed later. These processor elements display the context free stochastic property in which the future state of the system depends only on the present state of the system and not on the path by which the present state was achieved. As such, architecturally, the context free processor elements are uniquely different from conventional processor elements in two ways. First, the elements have no internal permanent storage or remnants of past events such as general purpose registers or program status words. Second, the elements do not perform any routing or synchronization functions. These tasks are performed by the software TOLL and are implemented in the LRDs. The significance of the architecture is that the context free processor elements of the present invention are a true shared resource to the LRDs.

Finally, the register set and condition code set files in contexts **660** can also be constructed of commonly available components such as AMD 29300 series register files, available from Advanced Micro Devices, 901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088. However, the particular configuration of the files **660** as shown in FIG. **6** is unique under the teachings of the present invention and will be discussed later.

The general operation of the present invention based upon the example set forth in Table **1** is illustrated with respect to the processor-context register file communication in FIGS. *7a*, *7b*, and *7c*. As mentioned, the time-driven control of the present invention is found in the addition of the extended intelligence relating to the logical processor number (LPN) and the instruction firing time (IFT) as specifically set forth in FIG. **4**. FIG. **7** generally represents the configuration of the context free processor elements PE0 through PE4 with registers R0 through R4, R10 and R11 of the register set and condition code set file **660**.

In explaining the operation of the TDA system architecture **600** for the single user example in Table 1, reference is made to Tables 3 through 5. In the example, for instruction firing time T16, the context-PE network **670** is set up to interconnect processor element PE0 with registers R0 and R10, processor element PE1 is interconnected with registers R1 and R11 processor element PE2 is interconnected with register R4. Hence, during time T16, the three processor elements PE0, PE1, and PE2 process instructions I0, I1, and I4 concurrently and store the results in registers R0, R10, R1, R11, and R4. During time T16, the LRD **620** selects and delivers the instructions that can fire during time T17 to the appropriate processor elements. During instruction firing time T17, only processor element PE0 which is now assigned to process instruction I2 and it is interconnected with registers R0, R1, and R2. The BEU is also connected to the condition codes (not shown). Finally, during instruction firing time T18, only processor element PE0 is interconnected to registers R2 and R3.

Several important observations need to be made. First, when a particular processor element (PE) places the results in a given register, any processor element, during a subsequent instruction firing time (IFT), can be interconnected to that register when performing a subsequent operation. For example, processor element PE1 for instruction I1 loads register R1 with the contents of a memory location during IFT T16 as shown in FIG. *7a*. During instruction firing time T17, processor element PE0 is now interconnected with register R1 to perform an additional operation on the results stored therein. Under the teachings of the present invention, each processor element (PE) is "totally coupled" to the necessary registers in the register file **660** during any particular instruction firing time (IFT) and, therefore, there is no need to move the data out of the register file for delivery to another resource; e.g. in another processor's register as in some conventional approaches.

In other words, under the teachings of the present invention, each process or element can be totally coupled, in any individual instruction firing time, to any one of the shared registers **660**. In addition, under the teachings of the present invention, none of the processor elements has to contend (or wait) for the availability of a particular register or for results to be placed in a particular register as is found in some prior art systems. Also during any individual firing time, any processor element has full access to any configuration of registers in the register set file 660 as if such registers were their own internal registers.

Hence, under the teachings of the present invention, the added intelligence as shown in FIG. 4, is based upon detected natural concurrencies within the object code. The detected concurrencies are analyzed by TOLL which logically assigns individual logical processor elements (LPNs) to process the instructions in parallel, and assigns unique firing times (IFTs) so that each processor element (PE) for its given instruction will have all necessary resources available for processing according to its instruction requirements. In the above example, the logical processor numbers correspond to the actual processor assignment or LPND to PED, LPN1 to PE1, LPN2 to PE2, and LPN3 to PE3. The invention is not so limited since any order such as LPN0 to PE1, LPN1 to PE2, etc. could be used. Or, if the TDA system had only more or less than four processors, a different assignment could be used as will be discussed.

The timing control for the TDA system is provided by the instruction firing times—i.e., time-driven. As can be observed in FIGS. 7*a* through 7*c*, during each individual instruction firing time, the TDA system architecture composed of the processor elements 640 and the PE-register set file network 670, takes on a new and unique and particular configuration fully adapted for the individual processor elements to concurrently process instructions while making full use of all the available resources. The processor elements are context free since data, condition, or information relating to past processing is not required, nor does it exist internally to the processor element. The processor elements of the present invention react only to requirements of each individual instruction and are interconnected to the necessary shared registers.

4. Summary

In summary, the TOLL software 110 for each different program or compiler output 100 de-analyzes the natural concurrencies existing in each single entry, single exit (SESE) basic block (BB) and adds intelligence comprising a logical processor number (LPN) and an instruction firing time (IFT) to each instruction. In a MIMD system of the present invention as shown in FIG. 6, each context would contain a different user executing the same or different programs. Each user is assigned a different context and as shown in FIG. 7, the processor elements (PEs) are capable of individually accessing the necessary resources such as registers and condition codes storage required by the instruction. The instruction itself carries the shared resource information (i.e., registers and condition code storage). Hence, the TOLL software statically allocates only once for each program the necessary information for controlling the processing of the instruction in the TDA system architecture in FIG. 6 to insure a time-driven decentralized control wherein the memory, the logical resource drivers, the processor elements, and the context shared resources are totally coupled through their respective networks in a pure, non-blocking fashion. The logical resource drivers (LRDs) are receptive of the basic blocks formed in an execution set and are responsible for delivering the instructions to the processor element 640 on a per instruction firing time (IFT) basis. While the example shown in FIG. 7 is a simplistic representation for a single user, it is to be expressly understood that the delivery by the logical resource driver 620 of the instructions to the processor elements 640, in a multi-user sense, makes full use of the processor elements as will be fully discussed subsequently. Because the timing and the identity of the shared resources and the processor elements are all contained within the extended intelligence added to the instructions by the TOLL software, each processor element 640 is context free and, in fact, from instruction firing time to instruction firing time can process individual instructions of different users and their respective context. As will be explained, in order to do this, the logical resource driver 620, in a predetermined order, deliver the instructions to the processor element 640 through the PE-LRD network 650.

DETAILED DESCRIPTION

1. Detailed Description of Software

In FIGS. 8 through 11, the details of the TOLL software 110 of the present invention are set forth. In FIG. 8, the conventional output from a compiler is delivered to the TOLL software at the start stage 800. The following information is contained within the conventional compiler output 800: (a) instruction functionality, (b) resources required by the instruction, (c) locations of the resources (if possible), and (d) basic block boundaries. TOLL software then starts with the first instruction at stage 810 and proceeds to determine "which" resources are used in stage 820 and to determine "how" the resources are used in stage 830. This continues for each instruction within the instruction stream through stages 840 and 850 and was discussed in the previous section.

When the last instruction is processed in stage 840, a table is constructed and initialized with the "free time" and "load time" for each resource. Such a table is set forth in Table 7 for the inner loop matrix multiply example and at initialization contains all zeros. The initialization occurs in stage 860 and once constructed the TOLL software proceeds to start with the first basic block in stage 870.

TABLE 7

Resource	Load Time	Free Time
R0	T0	T0
R1	T0	T0
R2	T0	T0
R3	T0	T0
R4	T0	T0
R10	T0	T0
R11	T0	T0

In FIG. 9, the TOLL software continues the analysis of the instruction stream with the first instruction of the next basic block in stage 900. As stated previously, TOLL performs a static analysis of the instruction stream. Static analysis assumes (in effect) straight line code, i.e., each instruction is analyzed as it is seen in a sequential manner. In other words, static analysis assumes that a branch is never taken. For non-pipelined instruction execution, this is not a problem, as there will never by any dependencies that arise as a result of a branch. Pipelined execution is discussed subsequently (although, it can be stated that the use of pipelining will only affect the delay value of the branch instruction).

Clearly, the assumption that a branch is never taken is incorrect. However, the impact of encountering a branch in the instruction stream is straightforward. As stated previously, each instruction is characterized by the resources (or physical hardware elements) it uses. The assignment of the firing time (and hence, the logical processor number) is dependent on how the instruction stream accesses these resources. Within TOLL, the usage of each resource is represented by data structures termed the free and load times for that resource. As each instruction is analyzed as it is seen, the analysis of a branch impacts these data structures in the following manner.

When all of the instructions of a basic block have been assigned firing times, the maximum firing time of the current basic block (the one the branch is a member of) is used to update all resources load and free times (to this value). When the next basic block analysis begins, the proposed firing time is then given as the last maximum value plus one. Hence, the load and free times for each of the register resources R0 through R4, R10 and R11 are set forth below in Table 8, for the example, assuming the basic block commences with a time of T16.

TABLE 8

Resource	Load Time	Free Time
R0	T15	T15
R1	T15	T15
R2	T15	T15
R3	T15	T15
R4	T15	T15
R10	T15	T15
R11	T15	T15

Hence, TOLL sets a proposed firing time (PFT) in stage 910 to the maximum firing time plus one of the previous basic blocks firing times. In the context of the above example, the previous basic blocks firing time is T15, and the proposed firing time for the instructions in this basic block commence with T16.

In stage 920, the first resource of the first instruction, which in this case is register R0 of instruction I0, is first analyzed. In stage 930 a determination is made as to whether or not the resource is read. In the above example, for instruction I0, register R0 is not read but written into and, therefore, stage 940 is next entered to make the determination of whether or not the resource is written. In this case, register R0 in instruction I0 is written into and stage 942 is entered. Stage 942 makes a determination as to whether the proposed firing time (PFT) for instruction I0 is less than or equal to the register resource free time for that resource. In this case, in Table 8, the resource free time for register R0 is T15 and, therefore, the instruction proposed firing time of T16 is greater than the resource free time of T15 and the determination is "no" and stage 950 is accessed.

The analysis by the TOLL software proceeds to the next resource which in the case for instruction I0 is register R10. This resource is both read and written by the instruction. Stage 930 is entered and a determination is made as to whether or not the instruction reads the resource. It does, so stage 932 is entered where a determination is made as to whether the current proposed firing time for the instruction (T16) is less than the resources load time (T15). It is not, so stage 940 is entered. Here a determination is made as to whether the instruction writes the resource - it does, so stage 942 is entered. In this stage a determination is made as to whether the proposed firing time for the instruction (T16) is less than the free time for the resource (T15). It is not, and stage 950 is accessed. The analysis by the TOLL software proceeds to the next resource which for instruction I0 is non-existent.

Hence, the answer to the determination of stage 950 is affirmative and the analysis then proceeds to FIG. 10. In FIG. 10, in stage 1000, the first resource for instruction I0 is register R0. The first determination in stage 1010 is whether or not the instruction reads the resource. As before, register R0 in instruction I0 is not read but written and the answer to this determination is "no" in which case the analysis then proceeds to stage 1020. In stage 1020, the answer to the determination as to whether or not the resource is written is "yes" and the analysis proceeds to stage 1022. Stage 1022 makes the determination as to whether or not the proposed firing time for the instruction is greater than the resource load time. In the example, the proposed firing time is T16 and with reference back to Table 8, the firing time T16 is greater than the load time T15 for register R0. Hence, the response to this determination is "yes" and stage 1024 is entered. In stage 1024, the resource load time is converted to the instructions proposed firing time and the table of resources updated to reflect that change. Likewise, stage 1026 is entered and the resource free time is updated to the instruction's proposed firing time plus one or T16 plus one equals T17.

Stage 1030 is then entered and a determination made as to whether there are any further resources used by this instruction. There are - register R10, and so analysis proceeds with this resource. Stage 1010 is entered where a determination is made as to whether or not the resource is read by the instruction. It is and so stage 1012 is entered where a determination is made as to whether the current proposed firing time (T16) is greater than the resources free time (T15). It is, so stage 1014 is entered where the resources free time is updated to reflect the use of this resource by this instruction. It is, and so stage 1022 is entered where a determination is made as to whether or not the current proposed firing time (T16) is greater than the load time of the resource (T15). It is, so stage 1024 is entered. In this stage, the resources load time is updated to reflect the firing time of the instruction, i.e., it is set to T16. Stage 1026 is then entered where the resource's free time is updated to reflect the execution of the instruction, i.e., it is set to T17. Stage 1030 is then entered where a determination is made as to whether or not this is the last resource used by the instruction. It is and stage 1040 is entered. The instruction firing time (IFT) is now set to equal the proposed firing time (PFT) of T16. Stage 1050 is then accessed which makes a determination as to whether or not this is the last instruction in the basic block which in this case is "no" and stage 1060 is entered to proceed to the next instruction, I1, which enters the analysis stage at A1 of FIG. 9.

In Table 9 below, that portion of the resource Table 8 is modified to reflect these changes. (Instructions I0 and I1 have been fully processed by TOLL.)

TABLE 9

Resource	Load Time	Free Time
R0	T16	T17
R1	T16	T17
R10	T16	T17
R11	T16	T17

The next instruction in the example is I1 and the identical analysis is had for instruction I1 for registers R1 and R11 as presented for instruction I0 with registers R0 and R10. Hence, Table 9, above, also shows the update of the resource table to reflect the analysis of instruction I1.

The next instruction in the basic block example is instruction I2 which involves a read of registers R0 and R1 and a write into register R2. Hence, in stage 910 of FIG. 9, the proposed firing time for the instruction is set to T16 (T15 plus 1). Stage 920 is then entered and the first resource in instruction I2 is register R0. The first determination made in stage 930 is "yes" and stage 932 is entered. At stage 932, a determination is made whether the instruction's proposed firing time of T16 is less than or equal to the resource register R0 load time of T16. It is important to note that the resource load time for register R0 was updated during the analysis of register R0 for instruction I0 from time T15 to time T16. The answer to this determination in stage 932 is that the proposed firing time equals the resource load time (T16 equals T16) and stage 934 is entered. In stage 934, the instruction proposed firing time is updated to equal the resource load time plus one or in this case T16 plus one equals T17. The instruction I2 proposed firing time is now updated to T17. Now stage 948 is entered and since instruction I2 does not write resource R0, the answer to the determination is "no" and stage 960 is entered to process the next resource which in this case is register R1.

Stage 960 causes the analysis to take place for register R1 and a determination is made in stage 930 whether or not the resource is read. The answer, of course, is "yes" and stage 932 is entered. This time the instruction proposed firing time is T17 and a determination is made whether or not the instruction proposed firing time of T17 is less than or equal to the resource load time for register R1 which is T16. Since the instruction proposed firing time is greater than the register load time (T17 is greater than T16), the answer to this determination is "no" and stage 940 is entered which does not result in any action and, therefore, the analysis proceeds to stage 950. The next resource to be processed for instruction I2 in stage 960 is resource register R2.

The first determination of stage 930 is whether or not this resource R2 is read. It is not and hence the analysis moves to stage 940 and then to stage 942. At this point in time the instruction I2 proposed firing time is T17 and in stage 942 a determination is made whether or not the instructions proposed firing time of T17 is less than or equal to resources, R2 free time which in Table 8 above is T15. The answer to this determination is "no" and therefore stage 950 is entered. This is the last resource processed for this instruction and the analysis continues in FIG. 10.

The analysis then proceeds to FIG. 10 and for instruction I2 the first resource R0 is analyzed. In stage 1010, the determination is made whether or not this resource is read and the answer is "yes." Stage 1012 is then entered to make the determination whether or not instruction I2 proposed firing time T17 is greater than the resource free time for register R0. In Table 9, the register free time for R0 is T17 and the answer to determination is "no" since both are equal. Stage 1020 is then entered which also results in a "no" answer transferring the analysis to stage 1030. Since this is not the last resource processed, stage 1070 is entered to advance the analysis to the next resource register R1. Precisely the same path through FIG. 10 occurs for register R1. Next, stage 1070 processes register R2. In this case, the answer to the determination of stage 1010 is "no" and stage 1020 is accessed. Since register R3 for instruction I2 is written, stage 1022 is accessed. In this case, the instruction I2's proposed firing time is T17 and the resource load time is T15 from Table 8, Hence, the proposed firing time is greater than the load time and stage 1024 is accessed. Stages 1024 and 1026 cause the load time and the free time for register R2 to be advanced, respectively, to T17 and T18 and the resource table is updated as shown in FIG. 10:

TABLE 10

Resource	Load Time	Free Time
R0	T16	T17
R1	T16	T17
R2	T17	T18

As this is the last resource processed, the proposed firing time of T17 becomes the actual firing time in stage 1040 and the next instruction is analyzed.

It is in this fashion that each of the instructions in the inner loop matrix multiply example are analyzed so that when fully analyzed the resource table appears in Table 11 below:

TABLE 11

Resource	Load Time	Free Time
R0	T16	T17
R1	T16	T17
R2	T17	T18
R3	T18	T19
R4	T16	T17
R10	T16	T17
R11	T16	T17

In FIG. 11, the TOLL software after performing the tasks set forth in FIGS. 9 and 10 enter stage 1100. Stage 1100 sets all resource free and load times to the maximum of those within the given basic block. For example, the maximum time set forth in Table 11 is T18 and, therefore, all free and load times are set to time T18. Stage 1110 is then entered to make the determination whether or not this is the last basic block for processing. If not, stage 1120 is entered to proceed with the next basic block and, if so, stage 1130 is entered and starts with the first basic block in the instruction stream. The purpose of this analysis is to logically reorder the instructions within each basic block and to assign logical processor numbers. This is summarized in Table 6 for the inner loop matrix multiply example. Stage 1140 performs the function of sorting the instruction in each basic block in ascending order using the instruction firing time (IFT) as the basis. Stage 1150 is then entered wherein the logical processor numbers (LPNs) are assigned. In making the assignment of the processor elements, the instructions are assigned as a set to the same instruction firing time (IFT) on a first come, first serve basis. For example, in reference back to Table 6, the first set of instructions for firing time T16 are I0, I1, and I4 are assigned respectively to processors PE0, PE1, and PE2. Next, during time T17, the second set of instructions I2 and I5 are assigned to processors PE0 and PE1, respectively. Finally, during the final time T18, the final instruction I3 is assigned to processor PE0. It is to be expressly understood that the assignment of the processor elements could be done in other fashions and is based upon the actual architecture of the processor element. As is clear, in the preferred embodiment the set of instructions are assigned to the logical processors on a first in time basis. After making the assignment, stage **1160** is entered to determine whether or not the last basic block has been processed and if not, stage **1170** brings forth the next basic block and the process is repeated until finished.

Hence, the output of the TOLL software results in the assignment of the instruction firing time (IFT) for each of the instructions as shown in FIG. **4**. As previously discussed, the instructions are reordered based upon the natural concurrencies appearing in the instruction stream according to the instruction firing times and, then, individual logical processors are assigned as shown in Table 6. While the above discussion has concentrated on the inner loop matrix multiply example, the analysis set forth in FIGS. **9** through **11** can be made on any SESE basic block (BB) in order to detect the natural concurrencies contained therein and then to assign the instruction firing times (IFTs) and the logical processor numbers (LPNs) for each user's program. This intelligence is then added to the reordered instructions within the basic block. This is only done once for a given program and provides the necessary time-driven decentralized control and processor mapping information to run on the TDA system architecture of the present invention.

The purpose of execution sets, as shown in FIG. **12**, is to optimize program execution by maximizing instruction cache hits within an execution set or, in other words, to statically minimize transfers by a basic block within an execution set to a basic block in another execution set. Support of execution sets consists of three major components: data structure definitions, pre-execution time software which prepares the execution set data structures, and hardware to support the fetching and manipulation of execution sets in the process of executing the program.

The execution set data structure consists of a set of one or more basic blocks and an attached header. The header contains the following information: the address **1200** of the start of the actual instructions (this is implicit if the header has a fixed length), the length of the execution set **1210** (or the address of the end of the execution set), and zero or more addresses **1220** of potential successor (in terms of program execution) execution sets.

The software to support execution sets manipulates the output of the post-compile processing which performs dependency analysis, resource analysis, resource assignment, and individual instruction stream re-ordering. The formation of execution sets uses one or of execution of basic blocks, and the grouping of basic blocks more algorithms for determining the probable order and frequency of execution of basic blocks, and the grouping of basic blocks accordingly. The possible algorithms are similar to the algorithms used in solving linear programming problems for least-cost routing. In the case of execution sets, cost is associated with branching. Branching between basic blocks contained in the same execution set incurs no penalty with respect to cache operations: it is assumed that the basic blocks of an execution set are resident in the cache in the steady state. Cost is associated with branching between basic blocks in different execution sets, because the target execution set's basic blocks may not be in cache. Cache misses delay program execution while the retrieval of the appropriate block from main memory to cache is made.

There are several possible algorithms which can be used to assess and assign costs under the teaching of the present invention. One algorithm is the static branch cost approach. Here one begins by placing basic blocks into execution sets based on block contiguity and the maximum allowable execution set size (this would be an implementation limit, such as maximum instruction cache size). The information about branching between basic blocks is known and is an output of the compiler. Using this information, one calculates the "cost" of the resulting grouping of basic blocks into execution sets, based on the number of (static) branches between basic blocks in different execution sets. One can then use standard linear programming techniques to minimize this cost function, thereby obtaining the "optimal" execution set cover. This algorithm has the advantage of ease of implementation; however, it ignores the actual dynamic branching patterns during actual program execution.

Other algorithms could be used under the teachings of the present invention which provide better estimation of acutal dynamic branch patterns. One example would be the collection of actual branch data from a program execution, and re-grouping of basic blocks using weighted assignment of branch costs based on the actual inter-block branching. Clearly, this approach is data dependent. Another approach would be to allow the programmer to specify branch probabilities, after which the weighted cost assignment would be made. This approach has the disadvantages of programmer intervention and programmer error. Still other approaches would be based using parameters, such as limiting the number of basic blocks per execution set, and applying heuristics to these parameters.

The algorithms described above are not unique to the problem of creating execution sets. However, the use of execution sets as a means of optimizing instruction cache performance is novel. Like the novelty of pre-execution time assignment of processor resources, the pre-execution time grouping of basic blocks for maximizing cache performance is not found in prior art.

The final element required to support execution sets is the hardware. As will be discussed subsequently, this hardware includes storage to contain the current execution set starting and ending addresses and to contain the other execution set header data. The existence of execution sets and the associated header data structures are, in fact, transparent to the actual instruction fetching from the cache to the processor elements. The latter depends strictly upon the individual instruction and branch addresses. The execution set hardware operates independently of instruction fetching to control the movement of instruction words from main memory to the instruction cache. This hardware is responsible for fetching basic blocks of instructions into the cache until either the entire execution set resides in cache or program execution has reached a point such that a branch has occurred to a basic block outside the execution set. At this point, if the target execution set is not resident in cache, the execution set hardware begins fetching the target execution set's basic blocks.

In FIG. **13**, the structure of the register set file of context zero of the contexts **660** is set forth. As shown in FIG. **13**, there are L levels of register sets with each register set containing N separate registers. For example, N could equal **31** for a total of 32 registers. Likewise, the L could equal **15** for a total of 16 levels. Note that these registers are not shared between levels; i.e.

each levels' set of registers is physically distinct from each other level.

Each level of registers corresponds to the registers available to a subroutine instantiation at a particular depth relative to the main program. In other words, level zero corresponds to the set of registers available to the main program, level one to any subroutine that is called directly from the main program. Level two corresponds to any subroutine called directly by a first level subroutine, level three to any subroutine called directly by a level two subroutine and so on.

As these sets of registers are independent, the number of levels corresponds to the number of subroutines that can be nested before having to physically share any registers between subroutines; i.e. before having to flush any registers to memory. The register sets in their different levels constitute a shared resource of the present invention and significantly saves system overhead in subroutine calls in that only rarely do sets of registers need to be pushed onto a stack in memory.

Communication between different levels of subroutines takes place in the preferred embodiment by allowing each routine three possible levels from which to obtain a register; the current level, the previous (calling) level and the global (main program) level. The designation of which level is to be accessed uses the static SCSM information attached to the instruction by the TOLL software. This can be illustrated by a subroutine call for a SINE function that takes as its argument a value representing an angular measure and returns the trigonometric SINE of that measure. This is set forth in Table 12:

TABLE 12

Main Program	Purpose
LOAD X, R1	Load X from memory into Reg R1 for parameter passing
CALL SINE	Subroutine Call - Returns result in Reg R2
LOAD R2, R3	Temporarily save results in Reg R3
LOAD Y, R1	Load Y from memory into Reg R1 for parameter passing
CALL SINE	Subroutine Call - Returns result in Reg R2
MULT R2, R3, R4	Multiply Sin (x) with Sin (y) and store result in Reg R4
STORE R4, Z	Store final result in memory at Z

The SINE subroutine is set forth in Table 13:

TABLE 13

Instruction	Subroutine	Purpose
I0	Load R1(10), R2	Load Reg R2, level 1 with contents of Reg R1, level 0
Ip-1	(Perform SINE), R7	Calculate SINE function and store result in Reg R7, level 1
Ip	Load R7, R2(10)	

Hence, under the teachings of the present invention and with reference to FIG. 14, instruction I0 of the subroutine loads register R1 of the current level (the subroutine's level or called level) with the contents of register R2 from the previous level (the calling routine or level). Note that the subroutine has a full set of registers with which to perform the processing independent of the calling routines register set. Upon completion of the subroutine call, instruction Ip causes register R7 of the current level to be stored into register R2 of the calling routines level (which returns the results of the SINE routine back to the calling program's register set).

The transfer between the levels occurs through the use of the SCSM statically provided information which contains the current procedural level of the instruction (i.e., the called routine or level), the previous procedural level (i.e. the calling routine or level) and the context identifier. The context identifier is only used when processing a number of programs in a multiuser system. This is shown in Table 13 for register R1 (of the calling routine) as R1(10) and for register R2 as R2(10). Note all registers of the current level have appended an implied (00) signifying current procedural level.

This differs substantially from prior art approaches where physical sharing of registers occurs between registers of a subroutine and its calling routine. The limiting of the number of registers that are available for use by the subroutine requires more system overhead for storing registers in memory. See, for example, the MIPS approach as set forth in "Reduced Instruction Set Computers" David A. Patterson, Communications of the ACM, January, 1985, Vol. 28, #1, Pgs. 8-21. In that reference, the first sixteen registers are local registers to be used by the subroutine, registers 16 through 23 are shared between the calling routine and the subroutine, and registers 24 through 31 are shared between the global (or main) program and the subroutine. Clearly, out of 32 registers that are accessible by the subroutine, only 16 can be privately used by the subroutine in the processing of its program. In the processing of complex subroutines, the remaining registers that are private to the subroutine may not (in general) be sufficient for the processing of the subroutine. Data shuffling (entailing the storing of intermediate data in memory) would occur resulting in significant overhead in the processing of the routine.

Under the teachings of the present invention, the transfers between the levels occur at compile time by adding the requisite information to the register identifiers as shown in FIG. 4, to appropriately map the instructions between the various levels. Hence, a completely independent set of registers are available to the calling routine and to each level of the subroutines. The calling routine, in addition to accessing its own complete set of registers, can also gain direct access to a higher set of registers using the aforesaid static SCSM mapping code added to the instruction as previously discussed. There is literally no reduction in the size of the register sets available to the subroutines as specifically found in prior art approaches. Furthermore, the mapping code for the SCSM information can be a field of sufficient length to access any number of desired levels. For example, a calling routine can access up to seven higher levels in addition to its own registers with a field of three bits. The present invention is not to be limited to any particular number of levels nor to any particular number of registers within a level. Under the teachings of the present invention, the mapping shown in FIG. 14 is a logical mapping and not a conventional physical mapping. For example, if three levels such as calling routine level, the subordinate level, and the global level, three bit maps are used: calling routine (00), subordinate level (01), and global level (11). Thus, each user's program is analyzed and the static SCSM window code added prior to the issuance of the user to a specific LRD. When the user is assigned to a specific LRD, the LRD dependent and dynamic SCSM information is added as it is needed.

2. Detailed Description of the Hardware

As shown in FIG. 6, the TDA system 600 of the present invention is composed of memory 610, logical resource drivers (LRD) 620, context free processor elements (PEs) 640, and shared context storage 660. The following detailed description starts with the logical resource drivers since the TOLL output is loaded into this hardware.

a. Logical Resource Drivers (LRDs)

The details of an individual logical resource driver (LRD) are set forth in FIG. 15. As shown in FIG. 6, each logical resource driver 620 is interconnected to the LRD-memory network 630 on one side and to the processor elements 640 through the PELRD network 650 on the other side. If the present invention were a SIMD machine, then only one LRD is provided and only one context is provided. For MIMD capabilities one LRD and context is provided for each user so that in FIG. 6 up to "n" users are shown.

The logical resource driver 620 is composed of the data cache section 1500 and an instruction selection section 1510. In the instruction selection section, the following components are interconnected. The instruction cache address translation unit (ATU) 1512 is interconnected to the LRD-memory network 630 over bus 1514. The instruction cache ATU 1512 is further interconnected over bus 1516 to an instruction cache control circuit 1518. The instruction cache control circuit 1518 is interconnected over lines 1520 to a series of cache partitions 1522*a*, 1522*b*, 1522*c*, and 1522*d*. Each of the cache partitions are respectively connected over busses 1524*a* 1524*b*, 1524*c*, and 1524*d* to the LRD-memory network 630. Each cache partition circuit is further interconnected over lines 1536*a*, 1536*b*, 1536*c*, and 1536*d* to a processor instruction queue (PIQ) bus interface unit 1544. The PIQ bus interface unit 1544 is connected over lines 1546 to a branch execution unit (BEU) 1548 which in turn is connected over lines 1550 to the PE-context network 670. The PIQ bus interface unit 1544 is further connected over lines 1552*a*, 1552*b*, 1552*c*, and 1552*d* to a processor instruction queue (PIQ) buffer unit 1560 which in turn is connected over lines 1562*a*, 1562*b*, 1562*c*, and 1562*d* to a processor instruction queue (PIQ) processor assignment circuit 1570. The PIQ processor assignment circuit 1570 is in turn connected over lines 1572*a*, 1572*b*, 1572*c*, and 1572*d* to the processor elements 640.

On the data cache portion 1500, the data cache ATU 1580 is interconnected over bus 1582 to the LRD-memory network 630 and is further interconnected over bus 1584 to the data cache control circuit 1586 and over lines 1588 to the data cache interconnection network 1590. The data cache control 1586 is also interconnected to data cache partition circuits 1592*a*, 1592*b*, 1592*c* and 1592*d* over lines 1593. The data cache partition circuits, in turn, are interconnected over lines 1594*a*, 1594*b*, 1594*c*, and 1594*d* to the LRD-memory

network 630. Furthermore, the data cache partition circuits 1592 are interconnected over lines 1596*a*, 1596*b*, 1596*c*, and 1596*d* to the data cache interconnection network 1590. Finally, the data cache interconnection network 1590 is interconnected over lines 1598*a*, 1598*b*, 1598*c*, and 1598*d* to the PE-LRD network 650 and hence to the processor elements 640.

The operation of each logical resource driver (LRD) 620 shown in FIG. 15 will now be explained. As stated previously, there are two sections to the LRD, the data cache portion 1500 and the instruction selection portion 1510. The data cache portion 1500 acts as a high speed data buffer between the processor elements 640 and memory 610. Note that due to the number of memory requests that must be satisfied per unit time, the data cache 1500 is interleaved. All data requests made to memory by the processor element 640 are issued on the data cache interconnection network 1590 and intercepted by the data caches 1592. The requests are routed to the appropriate data caches 1592 by the data cache interconnection network 1590 using the context identifier that is part of the dynamic SCSM information attached to each instruction by the LRD that is executed by the processors. The address of the desired datum determines which cache partition the datum resides in. If the requested datum is present (i.e., a cache hit occurs), the datum is sent back to the requesting processor element 640.

If the requested datum is not present, the address delivered to the cache 1592 is sent to the data cache ATU 1580 to be translated into a system address and this address is then issued to memory. In response, a block of data from memory (a cache line or block) is delivered into the cache partition circuits 1592)Vunder control 1586. The requested data that is resident in this cache block is then sent through the data cache interconnection network 1590 to the requesting processor element 640. It is to be expressly understood that this is only one possible design. The data cache portion is of conventional design and many possible implementations are realizable to one skilled in the art. As the data cache is of standard functionality and design, it will not be discussed further.

The instruction selection portion 1510 of the LRD consists of three major functions; instruction caching, instruction queueing and branch execution. The system function of the instruction cache portion 1510 is typical of any instruction caching mechanism. It acts as a high speed instruction buffer between the processors and memory. However, the current invention presents methods for realizing this function that are unique.

The purpose of the instruction cache 1510 is to receive execution sets from memory, place the sets into the caches 1522 and furnish the instructions within the sets on an as needed basis to the processor elements 640. As the system contains multiple independent processors elements 640, requests to the instruction cache are for a set of concurrently executable instructions. Again, due to the number of requests that must be satisfied per unit time, the instruction cache is interleaved. The set size ranges from none to the number of processors available to the user. The sets are termed packets, although this does not necessarily imply that the instructions are stored in a contiguous manner. Instructions are fetched from the cache on the basis of their instruction firing time (IFT). The next instruction firing time register contains the firing time of the next packet of instructions to be fetched. This register may be loaded by the branch execution unit of the context as well as incremented by the cache control unit when an instruction fetch has been completed.

The next IFT register is a storage register that is accessible from the context control unit and the branch execution unit. Due to its simple functionality, it is not explicitly shown. Technically, it is a part of unit **1518**, the instruction cache control unit, and is further buried in the control unit **1660**. The key point here is that the NIFTR is merely a storage register and does not necessarily perform a sophisticated function.

The instruction cache portion **1510** receives an execution set from memory over bus **1524** and, in a round robin manner, places instructions word into each cache partition, **1522*a*, 1522*b*, 1522*c*** and **1522*d***. In other words, each instructions in the execution set is delivered wherein the first instruction is delivered to cache partition **1522*a***, the second instruction to cache partition **1522*b***, the third instruction to cache partition **1522*c*** and the fourth instruction to cache partition **1522*d***. The next instruction is then delivered to cache partition **1522*a*** and so on until all of the instructions in the execution set are delivered into the cache partition circuits.

All the words delivered to the cache partitions are not necessarily stored in the cache. As will be discussed, the execution set header and trailer may not be stored. Each cache partition attaches a unique identifier (termed a tag) to all the information that is to be stored in that cache partition. This is used to verify that information obtained from the cache is indeed the information desired. When a packet of instructions is requested, each cache partition determines if the partition contains an instruction that is a member of the requested packet. If none of the partitions contain an instruction that is a member of the requested packet (i.e., a miss occurs), the execution set that contains the requested packet is requested from memory in a manner analogous to a data cache miss.

If a hit occurs (i.e., at least one of the partitions **1522** contain an instruction from the requested packet), the partition(s) attach any appropriate dynamic SCSM information to the instruction(s). The dynamic SCSM information which is attached to each instruction is important for multi-user applications. The dynamically attached SCSM information identifies the context, n, of FIG. **6** assigned to a given user. Hence, under the teachings of the present invention, the system **600** is capable of delay free switching among many user contexts without requiring a master processor or access to memory.

The instruction(s) are then delivered to the PIQ bus interface unit **1544** of the LRD **620** where it is routed to the appropriate PIQ buffers **1560** by the logical processor number (LPN) contained in the extended intelligence that the TOLL software attached to the instruction. The instructions in the PIQ buffer with **1560** are buffered up for assignment to the actual processor elements **640** which is performed by the PIQ processor assignment unit **1570**. The assignment of the physical processor elements is performed on the basis of the number of processor elements currently available and the number of instructions that are available to be assigned. These numbers are dynamic. The selection process is set forth below.

The details of the instruction cache control **1560** of each cache partition **1522** of FIG. **15** are set forth in FIG. **16**. In each cache partition circuit **1522**, five circuits are utilized. The first circuit is the header route circuit **1600** which routes an individual word in the

header of the execution set over path **1520*b*** to the instruction cache control unit **1660**. The control of the header route circuit **1600** by the control unit **1660** is also over path **1520*b*** through the header path select circuit **1602**. The header path select circuit **1602** based upon the address received over lines **1502*b*** from the control unit **1660** selectively activates the required number of header routers **1600** in the cache partitions. For example, if the execution set has two header words, only the first two header route circuits **1600** are activated by the header path select circuit **1602** which causes the header information to be delivered over bus **1520*b*** to the control unit **1660** from the two activated header route circuits **1600**. As mentioned, each word in the execution set is delivered to each successive cache partition circuit **1552**.

Assume that the example of table 1 comprises an entire execution set and that appropriate header words appear at the beginning of the execution set. The instructions with the earliest instruction firing times (IFTs) listed first and with the lowest logical processor number first are:

TABLE 14

Header Word 1		
Header Word 2		
I0	(T16)	(PE0)
I1	(T16)	(PE1)
I4	(T16)	(PE2)
I2	(T17)	(PE0)
I5	(T17)	(PEI)
I3	(T18)	(PE0)

Hence, the example of Table 1 (i.e., the matrix multiply inner loop, now has associated with it two header words and the extended information of the firing time (IFT) and the logical processor number (LPN). As shown in Table 14, the instructions were reordered by the TOLL software according to firing times. Hence, as the execution set shown in Table 14 is delivered through the LRD-memory network **630** from memory, the first word is routed by partition CACHE0 to the control unit **1660**. The second word is routed by partition CACHE1 to the control unit **1660**, instruction I0 is delivered into partition CACHE2, instruction I1 into partition CACHE3, instruction I2 into partition CACHE0, and so forth. As a result, the caches partition **1522** now contain the instructions as shown in Table 15:

TABLE 15

Cache0	Cache1	Cache2	Cache3
—	—	I0	I1
I4	I2	I5	I3

It is important to clarify, the above example has only one basic block in the execution set (i.e., a simplistic example). In actuality, an execution set would have a number of basic blocks.

The instructions are then delivered into a cache random access memory (RAM) **1610** resident in each cache for storage. Each instruction is delivered from the header router **1600** over a bus **602** into the tag attaching circuit **1604** and then over line **1606** into the RAM 1610. The tag attacher circuit **1604** is under control of a tag generation circuit **1612** and is interconnected therewith over line **1520c**. Cache RAM **1610** could be a conven-